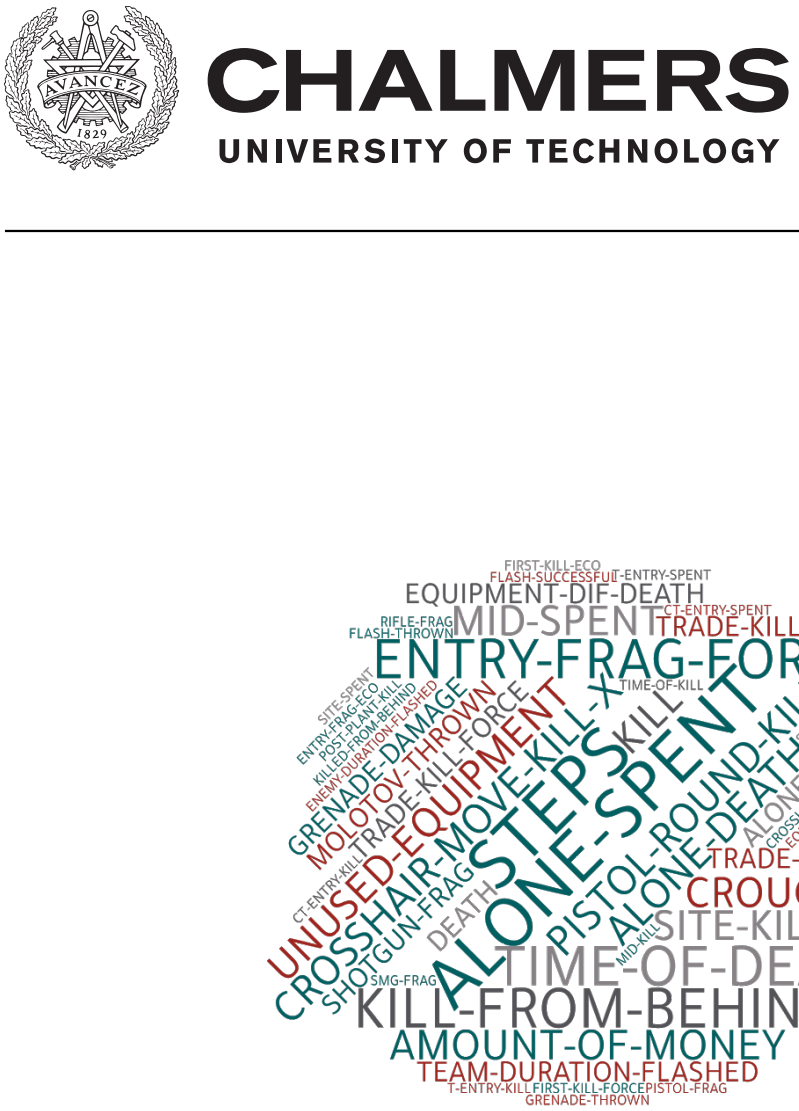

Predicting the outcome of CS:GO games using machine learning

Arvid Björklund, Fredrik Lindevall, Philip Svensson, William Johansson Visuri

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2018

BACHELOR OF SCIENCE THESIS DATX02-18-11

Predicting the outcome of CS:GO games using machine learning

Arvid Björklund
Philip Svensson
Fredrik Lindevall
William Johansson Visuri

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Predicting the outcome of CS:GO games using machine learning

Supervisor: Mikael Kågebäck
Examiner: Peter Damaschke

Bachelor of Science Thesis DATX02-18-11
Department of Computer Science and Engineering Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2017

Abstract

This work analyzes the possibility of predicting the result of a *Counter Strike: Global Offensive* (*CS:GO*) match using machine learning. Demo files from 6000 *CS:GO* games of the top 1000 ranked players in the EU region were downloaded from *FACEIT.com* and analyzed using an open source library to parse *CS:GO* demo files. Players from the matches were then clustered, using the *k*means algorithm, based on their style of play. To achieve stable clusters and remove the influence of individual win rate on the clusters, a genetic algorithm was implemented to weight each feature before the clustering. For the final part a neural network was trained to predict the outcome of a *CS:GO* match by analyzing the combination of players in each team. The results show that it is indeed possible to predict the outcome of *CS:GO* matches by analyzing the team compositions. The results also show a clear correlation between the number of clusters and the prediction accuracy.

Keywords: Video games, Esports, Competitive gaming, CS:GO, Counter-Strike, Machine learning

Contents

1 Introduction **1**
- 1.1 Purpose and goal . . . 1
- 1.2 Related Work . . . 1
  - 1.2.1 Sabermetrics . . . 2
  - 1.2.2 Ranking Systems . . . 2
    - 1.2.2.1 Elo Ranking System . . . 2
    - 1.2.2.2 *CS:GO*'s Ranking System: Glicko-2 . . . 3
  - 1.2.3 *CS:GO* Analyzing . . . 3
- 1.3 Scope . . . 4

2 Background **5**
- 2.1 *CS:GO* . . . 5
  - 2.1.1 Existing Roles . . . 5
- 2.2 Data collection . . . 6
  - 2.2.1 FACEIT . . . 6
  - 2.2.2 Web Scraping . . . 7
- 2.3 CS:GO Replay Parser . . . 7
  - 2.3.1 Parallel Computing . . . 7
- 2.4 Clustering Algorithm . . . 9
  - 2.4.1 K-means Clustering . . . 9
  - 2.4.2 The *k-value* . . . 9
  - 2.4.3 Weighting features . . . 10
- 2.5 Evolution Algorithm . . . 11
- 2.6 Neural network . . . 11
  - 2.6.1 Overview of the algorithm . . . 12
  - 2.6.2 Cost function . . . 13
  - 2.6.3 Activation function . . . 14
  - 2.6.4 Data partitions . . . 15
  - 2.6.5 Gradient descent . . . 15
  - 2.6.6 Stochastic gradient descent . . . 16
  - 2.6.7 Backpropagation . . . 16

3 Method **19**
- 3.1 Structure . . . 19
- 3.2 Collecting a data set . . . 20
  - 3.2.1 What data to analyze . . . 20
  - 3.2.2 Extracting features through the demo parser . . . 21
  - 3.2.3 Parser code example . . . 21
- 3.3 Clustering players into classes . . . 22

3.3.1 Stable clusters and weighting . . . 22
3.4 Predicting the result of CS:GO games . . . 23
3.4.1 Training data . . . 24
3.4.2 Validation data . . . 24
3.4.3 Test data . . . 24
3.4.4 Benchmark prediction . . . 24
3.5 Testing the program . . . 25
3.5.1 First test . . . 25
3.5.2 Second test . . . 25
3.5.3 Third test . . . 25
3.5.4 Fourth test . . . 26

4 Results and Discussion **27**
4.1 Weight fitness . . . 27
4.2 Benchmark prediction results . . . 29
4.3 Prediction accuracy . . . 32

5 Conclusion **33**
5.1 Future work . . . 33
5.1.1 The parser . . . 33
5.1.2 The data size . . . 33
5.1.3 More features . . . 34
5.1.4 Removing clustering . . . 34
5.1.5 Other games . . . 34
5.1.6 Optimization of the Neural Network . . . 35
5.2 Features tracked . . . i

Glossary

eSport: Competitions where people compete in video games.

CS:GO: An acronym for *Counter Strike: Global Offensive*, a multiplayer first person shooter game.

Flash grenade: A grenade in the game *CS:GO* used to temporarily blind and deafen players.

Demo file: A replay file of a *CS:GO* match which contains most of the information from the match.

Features: Used as a term for the dimensions that the a clustering algorithm takes as input. The different features in this project are actions that players do that are tracked.

1 Introduction

With the rapid advances made to computing power today, many new interesting areas of science have appeared. Competitive gaming is one of these areas as of the rising popularity and ease of retrieving a large amount of information that is already digitalized. Professional eSport teams are constantly looking to improve and one of the most important aspects of a successful organization is having a team roster that synergizes well. Today there is no concrete way of utilizing computers to evaluate how well a certain combination of players would perform in a given team. This product aims to solve this through analyzing individual players play style and grouping them into teams via machine learning, the product then evaluates these teams and returns which one, when matched against each other, is most likely to win.

1.1 Purpose and goal

The purpose of this project is to investigate if it is possible through the use of machine learning to verify that a *CS:GO* team composition is better than others without taking the individual players rank into account.

The goal is to create a program that can predict the chance of a given *CS:GO* team winning against another team. This should be done by first gathering data about different players and cluster them into groups based on how their style of play differ from each other. The clustering should not be based on their win-loss history or statistics that directly correlate with this. Finally, a neural network will be trained to take two teams as input and calculate the percentage chance of them winnings against each other. Thus finding out which of the teams that have the combination of players resulting in a higher win rate.

1.2 Related Work

Evaluating players based on key factors and statistics have previously been used in sports, one famous example is the concept Sabermetrics described below. Even in *CS:GO*, some research has been done into finding the optimal play in certain situations.

Currently, there also exists many different methods and algorithms to rank and match players, in many different sports, of similar skill level. In *CS:GO*, these ways of ranking players are mainly used when matching similarly skilled players.

1.2.1 Sabermetrics

Billy Beane, the general manager of baseball team Athletics in 1997-2015, popularized the famous statistics analyzing method Sabermetrics [1]. This method is based on collecting relevant data from the in-game activity of players. Statisticians would measure and analyze important numbers that could be relevant in evaluating player performances. Using these evaluations, players are thoroughly researched so that they can be put into a team to fill a specific function.

A similar philosophical basis to Sabermetrics is used as inspiration, collecting and analyzing data from in-game that is deemed relevant for the clustering to be based on.

1.2.2 Ranking Systems

Today, there exists technology that is used to compose *CS:GO* teams up to an optimal level [2]. These technologies are mainly used for matching similarly skilled players when searching for games versus unknown opponents. They differ from this projects way of determining what a good team composition is since they divide players purely by how much they win or lose instead of matching players that complement each other's skill. As stated earlier in Section 1.1, the goal is to group players, by clustering, into roles based on key factors and how well they execute these key factors instead of their win-loss history. When other ranking systems evaluate if a game is evenly matched each team is given a score that can be translated into a chance that the given team will win against the other. This is very similar to what this project is trying to achieve. The ranking system that is most commonly used today is the Elo system. The official matchmaking system in *CS:GO* uses an extension of the Elo system, called the Glicko-2 system [3].

1.2.2.1 Elo Ranking System The Elo system was created by Árpád Élő in 1959 mainly for chess players that competed on a high level. He succeeded in 1970 when the World Chess Federation (*FIDE*) adopted it as their main ranking system [4]. It is one of the oldest ranking systems that is still used today in chess and other sports including, football, baseball, and hockey [5][6] [7][8]. Even in the video-game industry the Elo system is widely used today including in *CS:GO*[9][10][11].

It works by assigning each player a respective score that reflects their skill level. Then, comparing the two player's scores, the system takes away points (from the loser) and gives points (to the winner) based on the difference in the player's scores. The loser loses

fewer points if they are the one with the lesser score, and wins more points if they were to win (since it was more unlikely for a lesser skilled player to win against a more skilled one). At the same time, the winner wins fewer points if they have a higher score (since it is expected for them to win against a lesser skilled opponent), and loses more points than usual if they have the higher score (since the one with the higher score "should" have won). This way, losing to a player with a higher score (i.e. more skilled) is not as punishing as losing to one with a lower score (i.e. less skilled). When used as a match predictor, it is quite straightforward; the player with a higher score has a higher chance of winning.

1.2.2.2 *CS:GO*'s Ranking System: Glicko-2

The game itself has an already implemented ranking system which places players into 18 Skill Groups, based on their performance in the game [12]. It was unclear which ranking system it was based on, as the company itself had never ushered a word around this topic to its players. In 2015, however, a company employee let out that *CS:GO* initially used the Glicko-2 ranking system for matchmaking [13]. Although throughout the years, the system has long been improved and adapted to better fit the player base [14].

It is worth noting that the Glicko-2 - and similarly Elo - system was designed for two-player games in mind. Since online multiplayer video games, such as *CS:GO*, are often team-based and involve more than one player on each team, thus requiring a far more complex method when it comes to calculating skill levels and ranking players correctly.

1.2.3 *CS:GO* Analyzing

The features that are possible to retrieve from the *CS:GO* demo files are many and need to be analyzed to see if they are relevant for deciding player roles. The data collection company Sixteen Zero has been doing this since July 2017 [15]. They work with retrieving data from professional games and selling useful information about how the best teams succeed. For example, they have a data collection of most grenades thrown in the previous year, including ones on a more detailed level as well [16]. If a team wants to maximize the effectiveness of a grenade thrown from point A to another point B, they can, via clustering, find the best method and timing to throw it. In this project, similar data will be analyzed and collected. Although, instead of tracking how and when the grenade was thrown, the player is given a score value depending on how effective it was. A higher score is given for more effective grenades, and a lesser score is given for less effective ones. The score value is also assigned a certain category under each player; in this project, the score would

be assigned to the player's efficiency of using a flash grenade. Sixteen Zero's database of information gives insight as to which statistics are relevant to track.

1.3 Scope

Since the number of features the players can do differently that can be extracted from a five-versus-five 3-D shooter game is so large and diverse, it is impossible to analyze every aspect of the game given the time frame of the project. Due to this, the scope of the project will not be to create the optimal algorithm to predict the best combination of a *CS:GO* team. Instead, the objective will be to first create a prototype to explore the possibilities of there being a way to predict the likelihood of a given combination of players winning over another. The features collected will be subjectively chosen as this is the only way to initially find what might be relevant features (more on data collection in Section 3.2.1). There is also no intention of constructing any technology used from scratch. For instance, the library used to extract data from demo files is an open source project created by the internet alias StatHelix (more info of this program in Section 2.3) and some of the machine learning algorithms will be from coding libraries such as TensorFlow[17].

One of the most important factors in how well certain players perform together is the communication part. It is not hard to imagine that a team needs teamwork to play efficiently. Communication is especially vital in *CS:GO*, where important, split-second decisions need to be made in real-time. Due to the time window of this project, only in-game data and statistics will be analyzed and studied. External factors, such as team communication, eye tracking or strategical decisions, outside of the demo files will not be tracked.

2 Background

The following chapter introduces and explains the different tools, techniques, and information used for each component of the program.

2.1 *CS:GO*

CS:GO is a multiplayer first-person shooter video game. The game is played played 5 versus 5 were teams take turns play as defenders and attackers. The objective of the attackers is to infiltrate and take control over one of two areas on the map. After they have taken control over the area they can plant a bomb, this starts a timer of 35 or 40 seconds until the bomb explodes. The defending side now has to defuse the bomb before it explodes. If the bomb explodes the attackers win the round, if it is defused the defenders win it. If the players on one side are killed and not able to complete their objective they lose the round. The first team to win 16 rounds win the game.

Since teams consist of multiple players working together each individual is often assigned a specific role. Examples of different roles can be snipers, entry killers, or supports (more on roles in Section 2.1.1). These can be compared to roles in football, such as defenders, midfielders, or attackers. To kill people in the game, a player also has to be good at aiming; that is, be able to quickly place the cursor on the enemy and shoot. This requires skill as well as a good reaction time and can be compared to a football player with a good kick and aim. A team in soccer, however, cannot consist of players that are only good at kicking. Similarly in *CS:GO*, teams need to be made up of different roles. Popular team compositions today usually consist of two offensive players, one supportive player, one defensive player, and one player who tries to infiltrate enemy lines.

2.1.1 Existing Roles

Today, clearly defined roles exist thanks to extensive gameplay from professionals and their analysis on what is needed in a team. Dignitas, a professional *CS:GO* team, wrote an article describing these roles [18]:

- Supportive player; a player that spends money on utility grenades and limits the vision of enemy players through the use flash and smoke grenades.
- Entry fragger; a player that often gets the first kill. The first person that goes into a site and gathers information about the enemy positions.
- Rifler; all around good aimer. Often the player with most kills on the team.
- Sniper; divides the map through holding positions that are dangerous for the enemy to cross. This style of play typically suits a more defensive player.
- Lurker; breaks through enemy lines and tries to kill them from unexpected places.

These roles are important since their characteristics are what the features to be collected will be based on when clustering players into roles (see Section 3.3). These different roles also prove that players cannot all play the same in a team-based game like *CS:GO*. That is, each player has their respective responsibility to fulfill in the game to prevail.

2.2 Data collection

The quality and quantity of training data are essential for good results when working with machine learning[19]. Therefore it is very important to find a good source of data. In this particular project, the data needed was competitive games between highly ranked *CS:GO* players. To acquire this the website *FACEIT* was used to collect data.

2.2.1 FACEIT

FACEIT, founded in London 2012[20], is an independent platform for professional competitions within online multiplayer video games. *FACEIT* have its own leaderboards with the best players using their services. This is essential to ensure getting games from highly skilled players. By looking at the match history of the highest ranked players in Europe a large source of high-quality game data can be acquired. The process used for getting the download links and demos is called web scraping.

2.2.2 Web Scraping

Web scraping, also known as web harvesting or web data extraction, is a way to collect and extract data from a website. This can be done by using the hypertext transfer protocol (HTTP) or by using a manual bot that can visit a more complex website and extract data from different scripts and databases[21]. *FACEIT* uses JavaScript Object Notation (JSON) requests and has almost no static HTML, thus making it impossible to scrape the files using a fast HTTP-based web scraping script. Instead, a more complex way of reading the website and collecting information was needed.

To solve the web scraping problem, the Selenium chrome API extension for C#[22] was used. This allowed the program to open a site using a Google Chrome window, then read the JSON requests when the files were dynamically loaded. When the request containing the download link was read it was also written to a file containing all of the download links.

2.3 CS:GO Replay Parser

Since a major part of the project was data collection, a fast way to extract relevant data from *CS:GO* demo files was needed. This will be done using an open source program called DemoInfo [23] that plays through the demo files without displaying the match to the viewer. The program has two major functions. The first being able to detect events that are called when a certain action happens in the replay such as a kill occurs or a grenade is thrown. The second is the ability to extract general information about the game at any given tick. This information includes, for example, player's positions and their viewing angles. This allows for extraction of features and information directly from the *CS:GO* demo files. Which is later used in the *k*-means algorithm to group players depending on how they perform compared to others, as described in the *k*-means clustering Section 2.4.1. The demo parser is written in C#, therefore, the part of the program that collects all the data is also written in C#.

2.3.1 Parallel Computing

When working with machine learning the sample of data needs to be very big [19] as mentioned in the previous section. This means that the parser needs to be very efficient in its extraction of data. To achieve this the parser is programmed using concurrent pro-

gramming. The speedup of a program related to amounts of threads it runs on can be approximated using Amdahl's law [24]. The law can be described as:

$$\text{maximum speedup} = \frac{1}{\underbrace{(1-p)}_{\text{sequential part}} + \underbrace{p/n}_{\text{parallel part}}}$$

where the p is the part of the program that can be parallelized and n is the number of processes. Amdahl's law is a simple way of describing the speedup but is very effective at describing the reality. In Figure 1 a visual representation is given by how efficient concurrent programming can be.

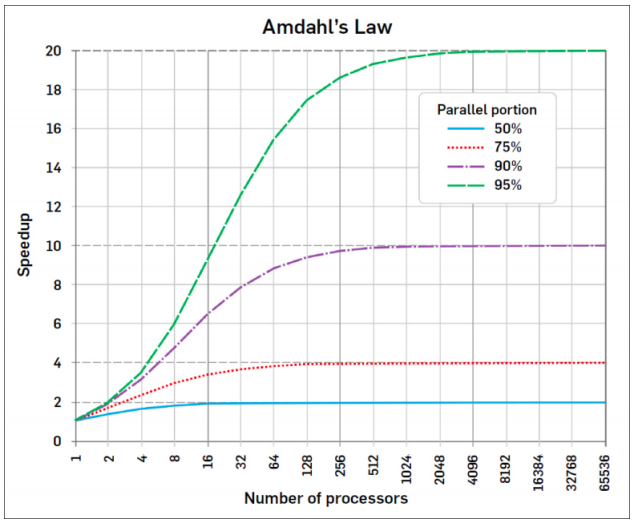

Figure 1: A graph of the potential speedup of a paralleled program where the x-axis is the number of processors and the y-axis is the number of times the program can potentially speed up. The different curves are how well paralleled the program is coded[25].

While the parser can be sped up in proportion to Amdahl's law the bottleneck in the program is downloading game replays. Thousands of replays are to be downloaded so the CPU handles these calls concurrently. Since the download speed is not throttled by the CPU the increase in speed is more linear. As long as the bandwidth is not exceeded each demo file downloading simultaneously will increase the speed of the program by a fraction. This formula for the increase is represented below.

$$S = \{1/n \,|\, n < \text{band with}\}$$

n is the number of concurrent downloads and S is the total time the parsing will take (*where S = 1 means it will take the full time and S = 0.5 means it will take half the time*). For the parsing to be efficient the amount of downloads needs to be different according to how much bandwidth is available. Therefore this can easily be changed with a variable within the program.

2.4 Clustering Algorithm

Clustering is the task of grouping a set of features in different clusters. The clusters should contain features with similar attributes. This technique is used in this project to group players with similar playstyles. There are a variety of ways to cluster features. The style that best fit this project was deemed to be strict partitioning with hard clustering. This means that a feature cannot be a part of several clusters and they have to belong to one cluster at least (i.e. they can not be without a cluster). The type of clustering called centroid models were able to cluster with these prerequisites. Therefore, a popular centroid algorithm called *k*-means was chosen [26]. This method of clustering was first mentioned in 1967 by James MacQueen [27].

2.4.1 K-means Clustering

Applying *k*-means on data is quite simple. The way it works is by first deciding a number *k*, which will represent how many clusters are in the output. (More on how to choose this value in Section 2.4.2.) *k* number of centroids are then randomly put in the space with data points. Each data point is then assigned to the closest centroid. The centroids then look at its assigned data point position in the space and averages where they are. This moves the centroid to the middle of its data points. The data points are then assigned the new closest centroid. This is then repeated until a stable point is found. A visualization can be seen in Figure 2.

2.4.2 The *k-value*

The *k*-value needs to be chosen before the data is seen. This means that it is important that the data is familiar before applying the algorithm. In the example seen in Figure 2 it is clear that there should be three or maybe four clusters to group correctly. This is easily seen because the data is only in two dimensions. The data used in this project will

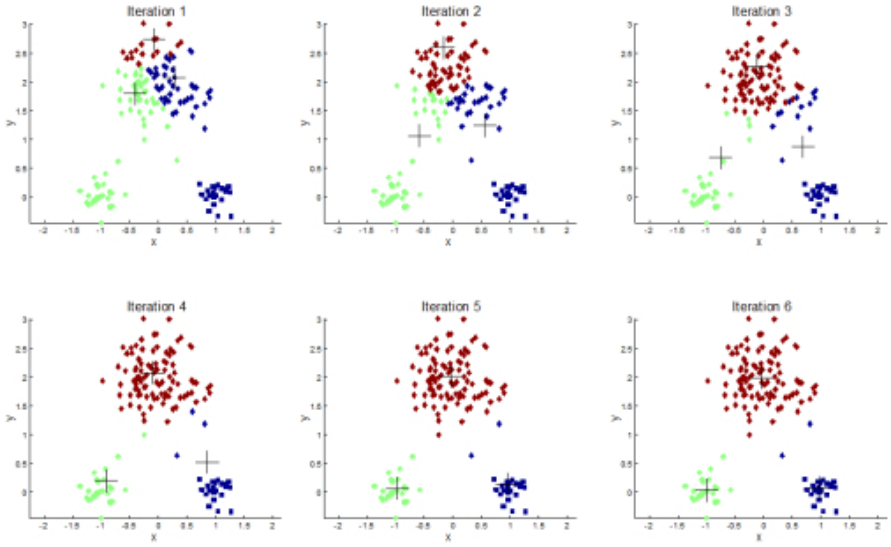

Figure 2: Iteration 1: A *k*-value of 3 is chosen and the centroids are randomly placed in the plane. Iteration 2: The centroids have been averaged a new position and assigned new data points. Iteration 3-5: The process is repeated. Iteration 6: The centroids have found the 3 clusters and a stable state has been found. [28]

be in around 50 dimensions and can therefore not as easily be plotted in such way. The way the *k*-value will be chosen initially in this project is using current roles in *CS:GO*. As can be read in Section 2.1.1. There are about 5 roles and that will be used as a reference point when choosing the initial value. This value will be changed to study the results and differences between other *k*-values.

2.4.3 Weighting features

In order to control the clustering outcome, a system to weight each feature in the clustering algorithm was implemented. The weighting assigns a number between 1 and 0 to each feature and when the centroids are placed the features with a low number have less impact on where the centroids are put. The outcome should be stable clusters and not randomly scattered over the data set. The winrate of a cluster should be as close to 50% as possible (for more information see Section 3.4.4). See Section 3.3.1 for further information on the implementation and how the weights were calculated.

2.5 Evolution Algorithm

Evolution Algorithm (EA) is a biologically inspired stochastic optimization algorithm. The basic concept is to use biologically inspired mechanisms such as reproduction, mutation, recombination to solve an optimization problem. The idea is to have an initial, randomly generated population of individuals that are sorted by their fitness. Fitness is the value of some function evaluating how good an individual solves the function you want to either minimize or maximize. High fitness equals a high chance of reproducing or surviving and low chance of dying. This combined with random mutation makes the algorithm very general.

Evolution Algorithms are great for approximating solutions for complex optimization problems[29]. It can be used in many different types of problems as long as it is possible to calculate the fitness of each individual in a population.

2.6 Neural network

A feedforward neural network was chosen for the last part of the project since the way it handled data seemed to be in line with what the program should achieve. The inputs of the neural network would be ten different roles which would compose two teams. The outputs would describe the chance of each team beating the other.
Consider the following problem:

Given a set of k input vectors, features, of dimension n

$$\mathbf{X}_i = \{x_1^i, ..., x_n^i\}, \quad i = 1, ..., k, \tag{1}$$

along with their corresponding output vectors, labels, of dimension m

$$\mathbf{Y}_i = \{y_1^i, ..., y_m^i\}, \quad i = 1, ..., k, \tag{2}$$

construct the function $f(\mathbf{X})$ that best maps the input $\mathbf{X}$ to the output $\mathbf{Y}$. The resulting function is going to be constructed as a nested function with p internal functions. This problem can be solved using a neural network, and is exactly what is going to be used in order to determine the output vector

$$Y = \{P(A\ wins),\ P(B\ wins)\} \tag{3}$$

from the input vector

$$X = \{Team\ Composition\ A,\ Team\ Composition\ B\}. \tag{4}$$

2.6.1 Overview of the algorithm

A neural network can be constructed in a number of different ways and there are many different types of neural networks with applications in different types of problems. A feedforward neural network will be constructed, which is one of the simpler types of neural networks. A feedforward neural network can be split up into three different parts, one input layer, one output layer and one or more hidden layers. Each of these layers consists of a number of nodes, called neurons, and is fully connected with weighted edges, called synapses, to the next layer. This is visualized in the following picture.

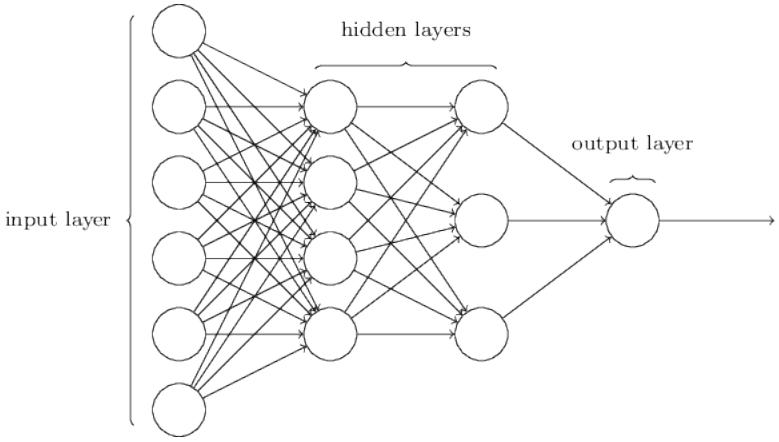

Figure 3: Visualization of a feedforward neural network

The simplified algorithm used to compute the function $f(X)$ is the following:

1. Randomize the weights of each synapse
2. Split the set of labelled input data into training and testing data
3. Insert one vector from the training data into the neurons of the input layer
4. Calculate each node in the next layer as a function of the linear combination of each node in the previous layer with their respectively connected synapses
5. Repeat step 3 until one reaches the output layer
6. Record the cost of this iteration, using one of the ways to measure cost described in section 2.6.2
7. Repeat steps 3-6 for all training data
8. Calculate the change in synapse weights that would minimize the total cost of the iterations above and apply this change to the network, this is called backpropagation and is explained in Section 2.6.7.
9. Repeat this for as many iterations, epochs, as one sees improvements to the cost and accuracy of the network.

2.6.2 Cost function

Given $\mathbf{X}_i$ and $\mathbf{Y}_i$ as above, the cost of a prediction is defined as the function

$$C_i(\mathbf{x}) = C(f(\mathbf{X}_i), \mathbf{Y}_i). \tag{5}$$

This function gives a numerical value on the deviation of the estimated solution from the expected solution for each input vector. Summing these costs together gives a total cost of the network. There are several different types of cost functions, for example, the mean squared error,

$$C(\mathbf{x}) = \frac{1}{k} \sum_{i=1}^{k} (\mathbf{Y}_i - f(\mathbf{X}_i))^2, \tag{6}$$

and also the binary cross-entropy defined as

$$C(\mathbf{x}) = -\frac{1}{k}\sum_{i=1}^{k}\left[y \ln f(\mathbf{X}_i) + (1-y)\ln(1-f(\mathbf{X}_i))\right], \tag{7}$$

where y is a binary value which is equal to 1 if the prediction is correct, and 0 if not. Binary cross-entropy was used in the project due to it being better for binary classification problems.

2.6.3 Activation function

Define the function $f(X)$ as a nested function of the form $g(h(...(X)))$, where each internal function manipulates the input data of that specific layer. These functions are called activation functions, and they are mainly used for two different reasons. They are used to set boundaries to the output of each layer, and also to increase the complexity of the function to make it non-linear.

One of the more common activation functions is called the logistic function and is defined as

$$f(x) = \frac{1}{1+e^{-x}}, \tag{8}$$

which outputs the real number line onto the interval $(0, 1)$. This activation function has a problem concerning the fact that as x tends to positive or negative infinity, the derivative of the function tends to zero, which is a problem and is explained further in Section 2.6.7.

Another example of an activation function is the rectified linear unit, often referred to as RELU. RELU is defined as the following function

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}, \tag{9}$$

which is often used as it is superior to the logistic function in the hidden layers of the network, which again, will be explained further in Section 2.6.7. Since RELU is a piecewise linear function one can approximate any form of complex relationship between the input and the output by adding them together.

2.6.4 Data partitions

In order to efficiently train and evaluate the neural network, it is important to properly split the data matrix $\{\mathbf{X}, \mathbf{Y}\}$ into training, testing and validation data. These all have different roles in the construction of the neural network.

The training data is the largest subset of data which is used to tune the synapse weights described in Section 2.6.7.

The validation data is an unbiased subset of data in the sense that it is not used during training and is used to train the so called hyper parameters such as the number of layers and neurons in each layer which are described more thoroughly in Section 2.6.7.

The testing data is then used to assess the level of overfitting on the training data. Overfitting is a problem that occurs when the neural network creates a model that fits exceptionally well to the training data but does not carry this over to new untrained data. Determining the level of overfitting is achieved by assuming that the training data follows the same probability distribution as the entire set of data, hence, a well fitted model for the training data should also fit the testing data equally well.

2.6.5 Gradient descent

Given a multi-variable function $f(\mathbf{x}) = f(x_1, ..., x_n)$, the gradient of f is defined as

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}. \tag{10}$$

This can be seen as the generalization of the derivative in higher dimensions. The inner product of an arbitrary vector $\mathbf{v}$ and the gradient of f, $\langle \nabla f, \mathbf{v} \rangle$ is maximized when the vectors are aligned in the same direction. This implies that the maximum increase in f can be achieved by traversing in the direction of the gradient. In the case of optimizing a feedforward neural network, one is interested in minimizing the cost function described in Section 2.6.2, and one way to accomplish this is to apply an iterative procedure called gradient descent. This algorithm works under the assumption one has some base vector $\mathbf{x}$ which one inserts into f given above. To minimize this function, one calculates the

gradient of the function and then calculates the new vector $\mathbf{x}$ as

$$\mathbf{x}_{i+1} = \mathbf{x}_i - t\nabla f(\mathbf{x}_i), \tag{11}$$

where t is a scaling factor for the length of the traversed vector, commonly known as the step length. After each iteration, the new gradient is calculated and the process repeats itself until the minimum is found to a tolerable margin of error. The step length is a key factor in minimizing the function as too high of a value on t makes convergence impossible, and too small of a value has the effect of requiring unreasonably many iterations before the minimum is found.

2.6.6 Stochastic gradient descent

Stochastic gradient descent is an algorithm that is very similar to the normal gradient descent presented above. With gradient descent, one is interested in calculating the gradient based on the full span of the input vectors, however, this is not computationally efficient when the input is large[30]. Stochastic gradient descent is an attempt to solve this problem by introducing noise as one calculates the gradient for randomized subsets of the input vectors.

2.6.7 Backpropagation

Denote the value, or activation, of the i:th neuron in the j:th layer of the k:th input vector as $a_i^{j,k}$, which in the input layer simply means $a_i^{1,k} = x_i^k$, or the k:th value of the input vector. Also denote the value, or weight, of the synapse connecting the i:th neuron in the j:th layer to the n:th neuron in the (j+1):th layer by $w_{i,n}^j$. Finally, denote the value of the bias of the i:th neuron in the j:th layer as b_i^j. Now, begin by describing a neural network. Each neuron's activation in any layer, except for the input layer, is determined by the inner product of each neuron's activation in the previous layer together with each connected weighted synapse. This is evaluated using the activation function f as follows,

$$a_i^{j+1,k} = f\left(\left\langle \mathbf{a}^{j,k}, \mathbf{w}_i^j \right\rangle + b_n^j\right) = f\left(a_1^{j,k}w_{1,i}^j + ... + a_n^{j,k}w_{n,i}^j + b_n^j\right) = f(s_i^{j,k}), \tag{12}$$

where n is the total number of neurons in the previous layer, $k = 1, ..., t$, that is, we have t input vectors, and b_n^j is the bias term which grants the option to increase or decrease the activation of each neuron regardless of the activations in the previous layer. Given a neural network with N layers and n neurons in each layer one is interested in describing how

the cost of the network is affected by the synapse weights of each layer in the network. In this example, using the definition of the cost function presented in Section 2.6.2, the following is given

$$C_k(\mathbf{x}) = C_k(\mathbf{a}^{N,k}, \mathbf{y}^k). \tag{13}$$

For example, in the case of using the mean squared error for the cost function it would be

$$C_k(\mathbf{x}) = (\mathbf{y}^k - \mathbf{a}^{N,k})^2, \tag{14}$$

where $\mathbf{y}^k$ denotes the expected output for the k:th input vector and $\mathbf{a}^{N,k}$ the observed output. Now, we are interested in how changing $w_{i,j}^{N-1}$ affects C_k for arbitrary neurons i and j, and in extension, how changing $w_{i,j}^{p}$ for $p = 1, ..., N-1$ affects C_k. It is also of interest to find how the bias b_j^p affects C_k as we are interested in minimizing the cost function by changing the synapse weights and neuron biases. That is, we are interested in finding

$$\nabla C_k = \begin{bmatrix} \frac{\partial C_k}{\partial w_{i,j}^1} \\ \frac{\partial C_k}{\partial b_j^1} \\ \vdots \\ \frac{\partial C_k}{\partial w_{i,j}^{N-1}} \\ \frac{\partial C_k}{\partial b_j^{N-1}} \end{bmatrix}, \tag{15}$$

in order to apply gradient descent. Begin by calculating

$$\frac{\partial C_k}{\partial w_{i,j}^{N-1}}. \tag{16}$$

By the chain rule, the relationship between how $w_{i,j}^{N-1}$ also changes $s^{N-1,k}$ which changes $a_j^{N,k}$ can be expressed as

$$\frac{\partial C_k}{\partial w_{i,j}^{N-1}} = \frac{\partial C_k}{\partial a_j^{N,k}} \frac{\partial a_j^{N,k}}{\partial s_j^{N-1,k}} \frac{\partial s_j^{N-1,k}}{\partial w_{i,j}^{N-1}}. \tag{17}$$

Looking at equation 14 it can be seen that in the case of using the mean squared error function the following is given

$$\frac{\partial C_k}{\partial a_j^{N,k}} = 2(a_j^{N,k} - y^k). \tag{18}$$

Equation 12 gives the following

$$\frac{\partial a_j^{N,k}}{\partial s_j^{N-1,k}} = f'(s_j^{N-1,k}), \tag{19}$$

and in the case of using the logistic function presented in equation 8 we get

$$\frac{\partial a_j^{N,k}}{\partial s_j^{N-1,k}} = \left(\frac{1}{1+e^{-s_j^{N-1,k}}}\right)' = \frac{e^{-s_j^{N-1,k}}}{\left(1+e^{-s_j^{N-1,k}}\right)^2} = f(s_j^{N-1,k})\left(1-f(s_j^{N-1,k})\right). \tag{20}$$

Finally, since

$$s_j^{N-1,k} = a_1^{N-1,k} w_{1,i}^{N-1,k} + ... + a_n^{N-1,k} w_{n,i}^{N-1,k}, \tag{21}$$

the following is given

$$\frac{\partial s_j^{N-1,k}}{\partial w_{i,j}^{N-1}} = a_j^{N-1,k}. \tag{22}$$

Now, looking back at equation 17, we get

$$\frac{\partial C_k}{\partial w_{i,j}^{N-1}} = 2(a_j^{N,k} - y^k) f(s_j^{N-1,k})\left(1-f(s_j^{N-1,k})\right) a_j^{N-1,k}. \tag{23}$$

Averaging the sum of these partial costs over k is used to give the final expression as

$$\frac{\partial C}{\partial w_{i,j}^{N-1}} = \frac{1}{t}\sum_{k=1}^{t} 2(a_j^{N,k} - y^k) f(s_j^{N-1,k})\left(1-f(s_j^{N-1,k})\right) a_j^{N-1,k}. \tag{24}$$

Now that an explicit expression is given for the derivative of the cost with respect to the previous layer's synapse weights the process can be repeated to go one layer further. To calculate the partial derivative of the cost function with respect to weights in previous layers the procedure is nearly identical, for example we have

$$\frac{\partial C_k}{\partial w_{i,j}^{N-2}} = \frac{\partial C_k}{\partial a_j^{N,k}} \frac{\partial a_j^{N,k}}{\partial s_j^{N-1,k}} \frac{\partial s_j^{N-1,k}}{\partial a_j^{N-1,k}} \frac{\partial a_j^{N-1,k}}{\partial s_j^{N-2,k}} \frac{\partial s_j^{N-2,k}}{\partial w_{i,j}^{N-2}}, \tag{25}$$

and in general the following formula holds

$$\frac{\partial C_k}{\partial w_{i,j}^{p}} = \frac{\partial C_k}{\partial a_j^{N,k}} \frac{\partial a_j^{N,k}}{\partial s_j^{N-1,k}} \frac{\partial s_j^{p,k}}{\partial w_{i,j}^{p,k}} \prod_{i=p+1}^{N-1} \left(\frac{\partial s_j^{i,k}}{\partial a_j^{i,k}} \frac{\partial a_j^{i,k}}{\partial s_j^{i-1,k}}\right). \tag{26}$$

3 Method

In order for the project to get started an early structure was specified. This helped with getting an overlooking picture of the project structure and to distribute the work between the group members. Since all the parts of the project were connected to each other it was important to know what needed to be completed in order for the next part of the program flow to work.

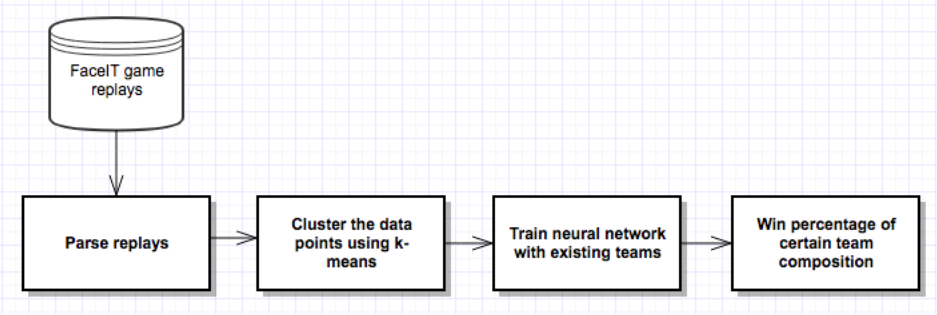

Figure 4: A vizualistion of the program process. From *FACEIT* replay servers to win percentage output.

3.1 Structure

The general structure of the project can be seen in Figure 4. The program uses web scraping to download the game replays from the *FACEIT* servers and process them in the parser to retrieve the features chosen to track. The data is then processed in the clustering algorithm. With every player clustered in a role, the neural network can then be trained on the games that have been analyzed. If a group of players always wins, the network sees that combination as a strong team composition and will value another team with similar playstyle high as well. When the network has been trained it will be able to take two teams of five players each and predict, with a certain accuracy, how well they would perform against each other. This is assuming the players have been clustered beforehand, if a player has not been clustered it will require 5-10 games in order to confidently place them in a cluster.

3.2 Collecting a data set

The data collection was solved using the techniques of web scraping described in Section 2.2.2. The web extraction worked in three different stages. The first step was to collect the names of the top 1000 ranked players in the EU region from the *FACEIT* website[31]. After collecting the players' names the program continued by storing the download links for the last 60-90 games played by each individual. This was done by taking all the links from each match room by scraping the players' profile pages.

After having all the links to each match room the web scraper continued by visiting all the match room links and there it found the links to each game demo file. This is a resource heavy process and takes a lot of time because of the JSON requests and the limitation on the server side for each request. This got fixed partly by threading the web scraping code and sending parallel JSON requests from different Google clients. This is a very costly way to solve the problem and needs a lot of computing power, therefore, the time for web scraping still is rather high. The speed is roughly 3000 game links each hour or 50 games each minute.

3.2.1 What data to analyze

A vital part of the project was to decide on what data to collect and to make sure it was relevant. *CS:GO* matches and existing teams were studied to find out what statistics were relevant and should be tracked. To avoid clustering players only based on their pure performance (as mentioned in Section 1.1), more features that did not directly correlate with players winning or losing are tracked. This meant tracking features were players would score points even if they did not perform on their top level. How a player would get kills were tracked, meaning that a players play-style now could depend on what types of weapon they prefers to use. Where on the map a player would spend most of their time was also tracked, this was particularly useful in order to find players that were good at defending/attacking different parts of the maps. A total of 50 features were tracked, some of which can be grouped in the following categories:

Statistics related to kills and deaths:

- Kills and deaths
- Type of weapon used when scoring a kill

- Where on the map a player got a kill

How players perform while playing at a disadvantage:

- Type of kills scored when a player has lower equipment value than their opponent
- Difference in money while scoring a kill

Others:

- Where players spend time on the map
- Position relative to other players
- Grenade usage
- Crosshair movement before scoring a kill

3.2.2 Extracting features through the demo parser

As mentioned in Section 2.3 the parser can extract a variety of data from a demo file. When features had been chosen methods to extract them could be implemented through the use of the parser.

3.2.3 Parser code example

For a better understanding of how the parser is used a code snippet from the data extraction is shown below.

```
parser.PlayerHurt += (sender, e) =>
    {
    if (!hasMatchStarted || e.Attacker == null ||
    e.Attacker.SteamID == 0 || e.Player.SteamID == 0)
        return;
    if (!playerData.ContainsKey(e.Attacker.SteamID))
        {
        playerData.Add(e.Attacker.SteamID, new
```

```
        PlayerData(e.Attacker.SteamID));
        }
    if (e.Weapon.Weapon.Equals(EquipmentElement.HE))
        {
        playerData[e.Attacker.SteamID].addNumber(parser.Map,
        PlayerData.STAT.GRENADE_DAMAGE,
        e.Attacker.Team, e.HealthDamage);
        }
};
```

The *parser.PlayerHurt* is an event that is called whenever a player receives damage. Each time this event occurs it is possible to extract information such as current time, amount of damage dealt, positions and players involved. In this example, the event is used to find out how much damage a player has done with grenades, over the entirety of the game. The two first *if statements* makes sure that the player is valid and exists in the tracking map. Sometimes during disconnects or bugs in general players may appear as null, resulting in null pointer exceptions. If the player does not exist the player gets added to the tracking system. The third if statement checks if the damage done was caused by a *High Explosive grenade*. If it did the *addNumber* function is called which adds the damage to an array associated with the player. These numbers are later distributed over the normal distribution curve and processed in the *k*means algorithm. Similar functions are done for all data chosen to be tracked.

3.3 Clustering players into classes

Seeing players as data points they can be clustered through *k*means (see Section 2.4) based on the features tracked, mentioned in Section 3.2.1.

3.3.1 Stable clusters and weighting

As mentioned in Section 1.1 players should not be clustered by how much they were winning or losing. This was required to prevent the neural network from looking at the win or loss percentage of a cluster when predicting a game result. As mentioned in Section 2.4.3 a weight system was implemented to control the clusters. The optimal set of weights was then calculated by an evolution algorithm (EA), the concept behind these algorithms is described in Section 2.5. The optimal weight would mean that the statistical evaluation (see Section 3.4.4) of which team would win a game should stay between 50-60%, while

the clusters still maintain a good stability i.e. the same clusters emerge every time the *k*means is run.

For the EA a genetic algorithm (GA) was used. This works by first creating a population full of individuals with random genes, in this case, the genes describe the different weights used in the clustering. which is the weights in this case. An individual solution is part of the set of all solutions that satisfy the constraints of the problem given. After that, the algorithm calculates the fitness of each individual. Having a higher fitness value means that the individual performs better in the fitness algorithm. Individuals with high fitness have a larger chance to reproduce and survive until next generation. If they have a low fitness there is a high chance of dying. To ensure differences and new genes in each generation mutation is implemented. This process is repeated over a fixed amount of generations or until the fitness of the best individual reach a chosen value.

The most important part of a GA is often the fitness function. If the algorithm does not have a way of evaluating which solution is most fit the result will not become optimal. For this optimization problem, the fitness of each generation is calculated by using the stability and win rate of each cluster. The individual with the best combination of stable clusters and clusters close to 50% win rate is most likely to pass on its genome. When calculating fitness the amount of features analyzed is also taken into consideration. This is done to ensure the use of as many features as possible when calculating the different points.

3.4 Predicting the result of CS:GO games

The feedforward neural network described in Section 2.6 was implemented using the tensorflow moduel keras and was used for the prediction part of the program. In order to construct the neural network properly, the data set collected needed to be split up into three different subsets, as described in Section 2.6.4. The data was split into

- Training data: 60%
- Test data: 20%
- Validation data: 20%

3.4.1 Training data

The training data consists of data from roughly 3600 *CS:GO* demo files. This data will be used in the clustering algorithm in order to find the players clusters and later to train the feedforward neural network. Since the network is rather simple the training will consist of weighting its synapses(edges) and input weights.

3.4.2 Validation data

The validation data consists of around 1200 demo files. This data is used to optimize the neural network.

3.4.3 Test data

After the validation data have been processed and the algorithm works as intended the product is going to be put to test. This will be done by testing previously unseen data. Using the classification of these players the program is going to attempt to predict the winning teams of each match. This is set aside to validate that the algorithm works as intended after the optimization and weighting. This means that none of the training is conducted on the validation data set.

3.4.4 Benchmark prediction

The benchmark prediction works as a benchmark for the neural network. The algorithm works by first calculating the win rate for each cluster in the data. It then tries to predict each result in every game by calculating the win rate of each side at an individual level. This is an information advantage compared to the neural network and is therefore a good comparison algorithm. When parsing fewer games the win rates were vastly different in each cluster and the algorithm scored an average prediction rate of around 70%. Given more data on each point results in an average performance, along with some weight optimization, the algorithm manages to score around 56-57%. If the neural network can beat this prediction percentage one can assume that optimal team compositions exist and that the players with highest win rates not always make up the statistically favoured team to win the game. This would strengthen the thesis that players can complement each other with key skills.

3.5 Testing the program

Since it was hard to get a program that would accurately predict game results in such a complex game as *CS:GO* the neural network and the types of data analyzed had to be tweaked and changed multiple times. During the project the results were continuously compared so that the project always was moving toward a final functional product. Around 4-5 large tests have been done during the project. More on the test results in Section 4. While doing tests as mentioned in Sections 3.5.1 - 3.5.4 the results of the project improved. These are the revelations that came up during the process.

3.5.1 First test

In the early stages of the project the program did not track many features since the main goal of the test was to see if the general structure was functional. Because of this the biggest changes to the next test was that more features were added. The program went from 15 to 30 features and this improved the prediction rate vastly. The conclusion from this test was mostly that the more features the better.

3.5.2 Second test

Even more features were added. Just like in the first test improvements were seen. A tendency for the program to cluster players into winning and losing clusters was seen though. This was not desirable since a team with players halving the highest win rate would be predicted to win, mostly disregarding the team composition.

3.5.3 Third test

More features were added again. A weighting system was added to combat the tendency for clusters with mostly winners or losers. This test was called modified weights (MW), or manually modified weights, and consisted of lowering the weight of the statistics thought to have the strongest correlation with winning or losing.

3.5.4 Fourth test

The parser was debugged so it crashed less. This resulted in a larger set of games that could be parsed. This larger set of data resulted in a more well trained neural network, which increased the prediction accuracy. The weighting process was also automated through an Evolution Algorithm (see Section 2.5). With these changes very distinct clusters with a decent prediction rate was achieved.

4 Results and Discussion

Below follows a presentation of the results and discussions of the project around them.

4.1 Weight fitness

The fitness of each feature weight set is determined by both the stability of the clusters (meaning that the clusters stay the same each time *k*means is run) and the win rate being as near 50% for each cluster. When comparing the different approaches it is clear that the weights that got calculated from the EA is superior in every way. The clusters are more stable as seen in Figure 5, when using only a few clusters the EA generated weights are roughly 18% more stable than all the weights being equal. When having more clusters the advantage gets less prominent but the EA constantly have most stability reaching over 99,5% stability when $k = 32$.

When it comes to the win rate and the prediction by the benchmark prediction method the EA outperforms both having all weights equal and the manually modified weights. As seen in the Figure 6. The difference between the different weight sets is minor but the EA is constantly better and in the best case when $k = 16$ the EA calculated weights lower the benchmark prediction with almost 2%.

The results of the manually modified weight set (see Section 3.5.3) is surprisingly worse than the equal feature overall. Lowering the kills and death feature weights makes the clustering less stable. This means that the clusters are less definable. When it comes to prediction the benchmark prediction performs worse both with k=16 and k=32 but the advantage is minor and the EA performs much better in both stability and lowering the benchmark prediction.

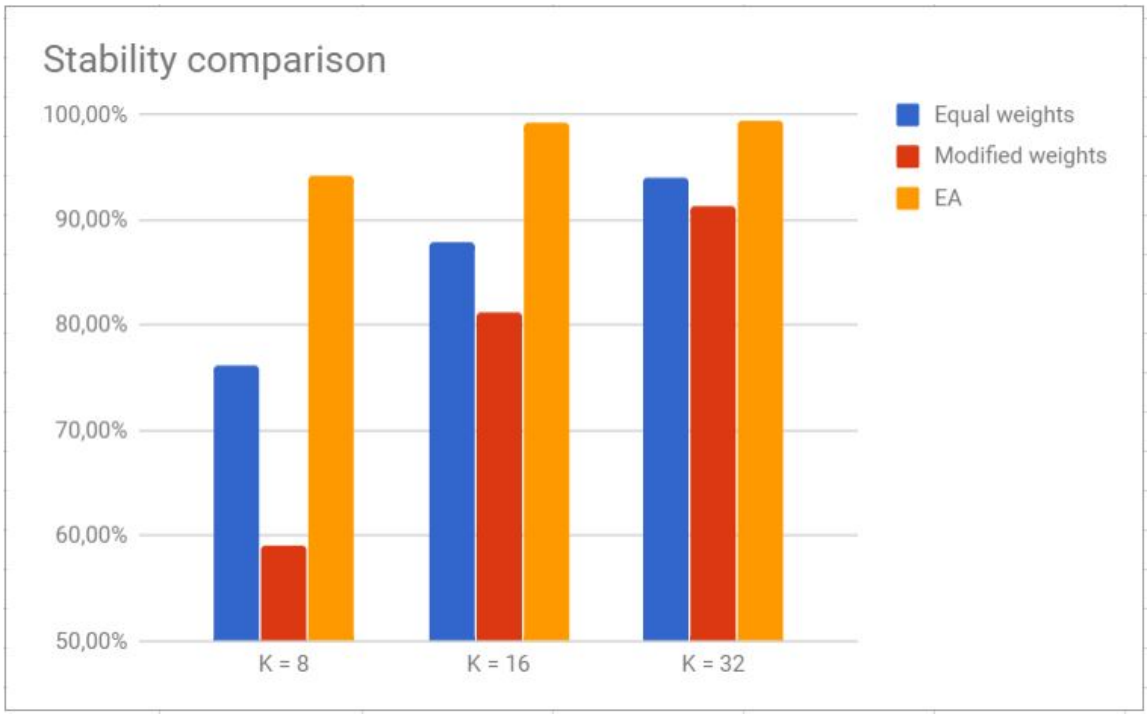

Figure 5: The graph above shows the stability of the different weight sets with weights set to 1 (Equal), modified weights (MW) (Section 3.5.3) and the weights calculated by the EA when using k = 8, 16 and 32

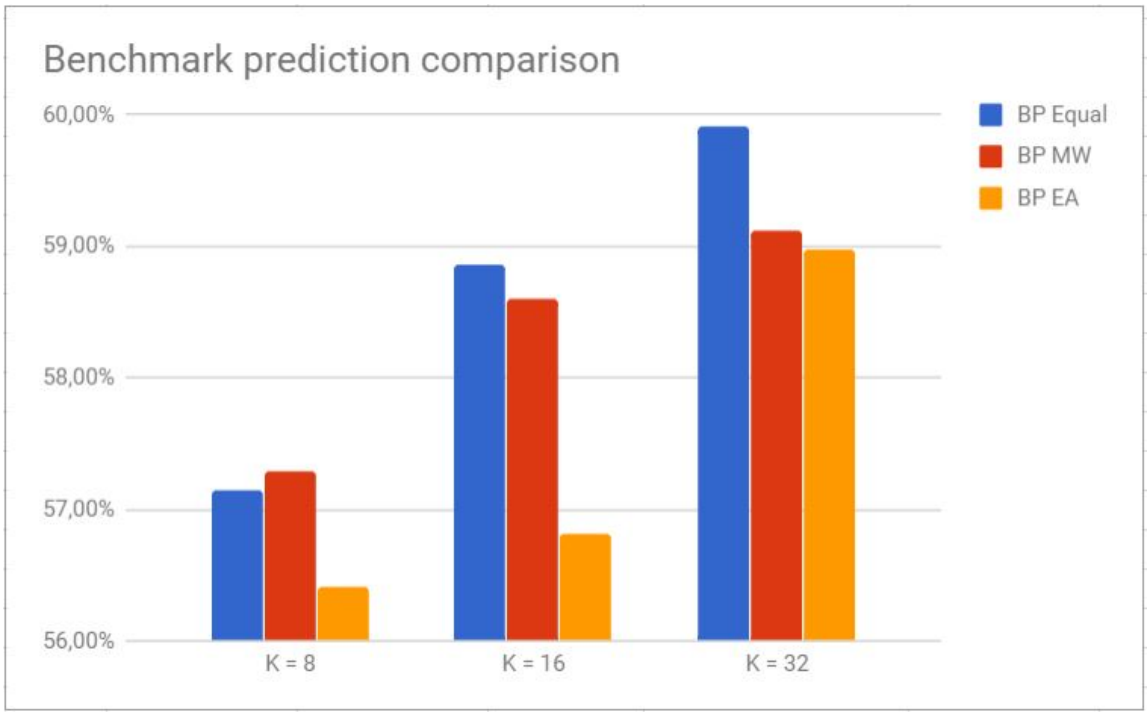

Figure 6: The graph above shows the prediction rate of the benchmark prediction (BP) method 3.4.4 with all weights set to 1 (Equal), modified weights (MW) (Section 3.5.3) and the weights calculated by the EA when using k = 8, 16 and 32

4.2 Benchmark prediction results

Because the training set is limited the data can be statistically predictable. Using a benchmark prediction function it can seen that the data have different win rate for different classes by default. The result for the statistical approach is nearest 50% when using the EA weights seen in Figure 6. An optimal result would be 50%.

Figure 7: Illustration of how the accuracy of the neural network improves with the number of epochs, excluding the testing data

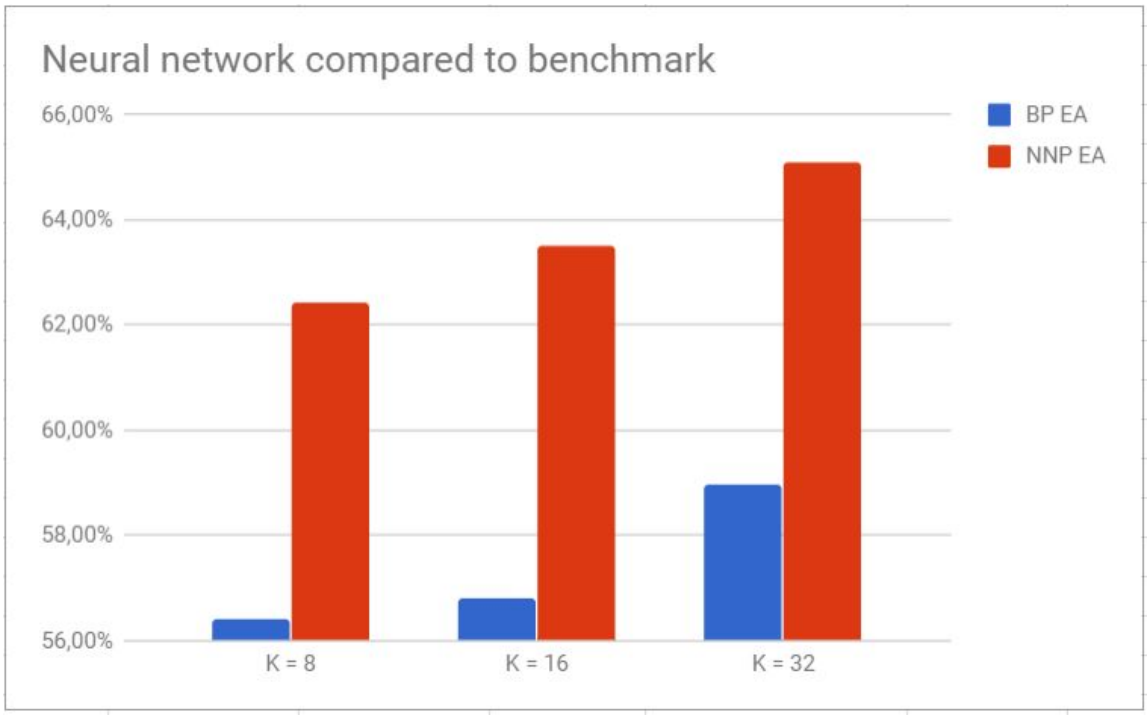

Figure 8: The graph above compares the prediction rate of the benchmark prediction (BP) 3.4.4 to the average neural network prediction (NNP) on the untrained test data. The tests are done with k = 8, 16 and 32.

4.3 Prediction accuracy

The neural network can consistently predict the outcome of games above the benchmark prediction method. This proves that the neural network is finding combinations of clusters that work well together and not only relying on the clusters win rates. All results are based off of the neural networks prediction on the test data. The network can predict the games with an accuracy of 65,11% compared to the benchmark prediction that relies only on win rate that gets a prediction of 58,97%. This can be seen in Figure 8. It is possible for the clustered data to cause the prediction to become inaccurate. This is due to the fact that the clusters does not take the entire span of each individual players properties into account. Instead, it bases its prediction off of the average player in their respective cluster, this is very hard to define and calculate. The benchmark prediction is a good baseline to compare the result against but not optimal. The margin of the results compared to the baseline is so big that the the conclusion of the project still is valid.

	8x Clusters					
	Stability Fitness	Win/Loss Fitness	Average Weight	Total Fitness	Benchmark Prediction	NN prediction
All weights 1	0,7621	0,8311	1	63,33%	57,15%	63,31%
Modified weight	0,5900	0,8107	0,9647	47,83%	57,29%	61,21%
EA	0,9423	0,8146	0,5635	76,76%	56,41%	62,43%
	16x Clusters					
	Stability Fitness	Win/Loss Fitness	Average Weight	Total Fitness	Benchmark Prediction	NN prediction
All weights 1	0,8792	0,7888	1	69,35%	58,85%	64,53%
Modified weight	0,8125	0,7917	0,9647	64,33%	58,60%	63,43%
EA	0,9934	0,8111	0,5635	80,58%	56,81%	63,52%
	32x Clusters					
	Stability Fitness	Win/Loss Fitness	Average Weight	Total Fitness	Benchmark Prediction	NN prediction
All weights 1	0,9405	0,7867	1	73,99%	59,92%	65,70%
Modified weight	0,9127	0,7965	0,9647	72,70%	59,12%	64,86%
EA	0,9951	0,7837	0,5635	77,99%	58,97%	65,11%

Figure 9: The results presented in one table. Organized in 8, 16 and 32 clusters. Each showing all weights 1, modified weights (MW) (Section 3.5.3) and weights calculated by the evolution algorithm (EA) (see Section 3.3.1). Stability fitness specifies how stable the cluster are, 1 being perfectly stable.

5 Conclusion

As seen in result the project purpose and goal was completed. The program was successfully able to predict the outcome of a CS:GO game by considering the team composition of each team. This is a big success for this area because it is a proof of concept. Since improvements were seen over all of the tests a conclusion can be drawn that the results can be improved upon with more time.

5.1 Future work

While the results were good there is room for improvements. These are the areas where the biggest improvements could be made.

5.1.1 The parser

The parser used in this project (described in Section 2.3) had several flaws. The one used to extract information from the demo files being open source was essential since it would be very time consuming to implement this during the project. However this also came with problems regarding bugs that was difficult to track, mostly due to the parser being created a couple of years ago and not much work has been done on it lately. It lacked features such as players line of vision or certain information regarding grenades. To truly collect all relevant features a new parser should be built from the ground up, similar to the one used by the company Sixteen Zero[15] mentioned in Section 1.2.3.

5.1.2 The data size

Even though the data size was big it could be bigger. Generally a bigger data size means a better learning process for the neural network. *FACEIT* only saves a certain amount of games for each player, and while the number of games available on the site at any time is very large (not a limitation for the scale of this project) more games would ensure a better training of the neural network. Although there were more replays that could be downloaded and parsed a faster parser and more computing power was needed.

5.1.3 More features

During the project about 50 features of the players were tracked. With this small amount of features it is hard to define more complex roles and play styles. This might make it hard to find out exactly what highly skilled players complement each other the best. With a new parser this could be changed. With more features the neural network would make a much more accurate prediction and the correlation between winning and losing would get even lower.

5.1.4 Removing clustering

The choice to cluster players was made in order for them to be seen as groups instead of individuals. The simple method of using *k*means with varying values of k worked very well for the problem given but a lot of information was lost between the clustering and the neural network. By this meaning that the data set before clustering contained thousands of individuals statistics. A lot of this information is then lost when the output of the algorithm compresses the data into groups. In the future it would be interesting to instead use a more complex neural network and have the game replays as a direct input. The downside of this approach is that it is harder to compose teams since the players will not be playing in their roles. The upside is that the prediction percentage may go way up since all the information that gets filtered by *k*-means makes the players less distinguishable from each other.

5.1.5 Other games

This project applied the machine learning evaluation on *CS:GO* mostly because the demo files from *CS:GO* were very easy to retrieve information from. But the same principle could be applied to almost any game. With an image recognition and information gathering from something like football the program could evaluate the best team in the same way. In fact aside from slight changes to the neural network the project could be used without alteration all the way from the *k*-means clustering algorithm (see Figure 4).

5.1.6 Optimization of the Neural Network

Due to time constraints, a decision was made not to conduct any in-depth optimization of the neural network's hyper parameters. This does not affect the final result in any significant way since the margin between the benchmark prediction and the neural network's prediction was more than sufficient to conclude the results that were needed. If a future project was exploring the possibilities of improving the prediction rates, this would be a very interesting area to explore.

References

[1] John T. Saccoman Gabriel B. Costa Michael R. Huber. *Understanding Sabermetrics: An Introduction to the Science of Baseball*. June 2008. URL: https://search.proquest.com/docview/225703156?pq-origsite=summon.

[2] Litvyakov B. Ignatov D.I. Makarov I. Savostyanov D. *Predicting Winning Team and Probabilistic Ratings in "Dota 2" and "Counter-Strike: Global Offensive" Video Games.* Dec. 2017. URL: https://doi-org.proxy.lib.chalmers.se/10.1007/978-3-319-73013-4_17.

[3] Mark E. Glickman. *Example of the Glicko-2 system*. Nov. 2013. URL: http://www.glicko.net/glicko/glicko2.pdf.

[4] A. E. Elo. *The rating of chess players past and present*. New York: Arco Publishing, 1978.

[5] Daniel Ross. *Arpad Elo and the Elo Rating System*. 2007. URL: https://en.chessbase.com/post/arpad-elo-and-the-elo-rating-system.

[6] Nate Silver. *We Are Elo?* June 2016. URL: http://baseballprospectus.com/article.php?articleid=5247.

[7] Bryan Cole. *Elo rankings for international baseball*. Aug. 2014. URL: http://www.beyondtheboxscore.com/2014/8/15/5989787/elo-rankings-for-international-baseball.

[8] Nate Silver. *Introducing NFL Elo Ratings*. Sept. 2014. URL: http://fivethirtyeight.com/datalab/introducing-nfl-elo-ratings.

[9] Blizzard Entertainment. *Welcome to Season 8 of competitive play*. Jan. 2018. URL: https://playoverwatch.com/en-us/blog/21363037.

[10] Golden Tee Fan. *Golden Tee Fan Player Rating Page*. July 2008. URL: http://www.goldenteefan.com/statistics/player-rating-handicap/.

[11] *Guild ladder*. July 2015. URL: http://wiki.guildwars.com/wiki/Guild_ladder.

[12] Wuzh. *Matchmaking*. Apr. 2018. URL: http://counterstrike.wikia.com/wiki/Matchmaking#Skill_groups.

[13] vitaliy_valve. *The Ultimate Guide to CSGO Ranking*. 2015. URL: https://www.reddit.com/r/GlobalOffensive/comments/2g3r4c/the_ultimate_guide_to_csgo_ranking/ckfhfir/.

[14] Henry Stenhouse. *CS:GO ranks, explained*. Feb. 2017. URL: https://www.pcgamer.com/csgo-ranks-explained/.

[15] *Sixteen zero homepage*. URL: http://sixteenzero.net/home/.

[16] Chris. *Most Efficient CT/T Flashes used on Overpass*. Jan. 2018. URL: http://sixteenzero.net/blog/2018/01/24/most-efficient-overpass-flashes/.

[17] *Tensorflow website*. URL: https://www.tensorflow.org/.

[18] Serenity. *Looking at Player Roles in CS:GO*. Aug. 2014. URL: http://team-dignitas.net/articles/blogs/CSGO/5750/Looking-at-Player-Roles-in-CSGO.

[19] Sahar Sohangir. Dingding WangAnna. PomeranetsTaghi M. Khoshgoftaar. *Big Data: Deep Learning for financial sentiment analysis*. Jan. 2018. URL: https://link.springer.com/article/10.1186%5C%2Fs40537-017-0111-6.

[20] *Faceit homepage*. URL: www.faceit.com.

[21] Eloisa Vargiu. Mirko Urru. *Exploiting web scraping in a collaborative filtering- based approach to web advertising*. 2013. URL: http://www.sciedu.ca/journal/index.php/air/article/view/1390.

[22] *Selenium web scraping software*. URL: https://www.seleniumhq.org/.

[23] Open source. *Demoinfo*. 2015-2018. URL: https://github.com/StatsHelix/demoinfo.

[24] Carlo A. Furia. *Introduction to concurrent programming*. 2016-2018. URL: http://www.cse.chalmers.se/edu/year/2017/course/TDA384_LP3/files/lectures/Lecture01-introduction.pdf.

[25] Keith Kirkpatrick. *Parallel Computational Thinking*. Dec. 2017. URL: https://cacm.acm.org/magazines/2017/12/223054-parallel-computational-thinking/fulltext.

[26] Saurav Kaushik. *An Introduction to Clustering and different methods of clustering*. Nov. 2016. URL: https://www.analyticsvidhya.com/blog/2016/11/an-introduction-to-clustering-and-different-methods-of-clustering/.

[27] James MacQueen. *Some methods for classification and analysis of multivariate observations*. 1967. URL: https://projecteuclid.org/euclid.bsmsp/1200512992.

[28] Andrew Pandre. *Cluster Analysis: see it 1st*. 2008. URL: https://apandre.wordpress.com/visible-data/cluster-analysis/.

[29] H.Mühlenbein. M.Gorges-Schleuter. O.Krämer. *Evolution algorithms in combinatorial optimization*. Apr. 1988. URL: https://www.sciencedirect.com/science/article/pii/0167819188900981.

[30] Léon Bottou. *Large-Scale Machine Learning with Stochastic Gradient Descent*. 2010. URL: http://leon.bottou.org/publications/pdf/compstat-2010.pdf.

[31] *Faceit ranking page*. URL: https://www.faceit.com/en/csgo/rankings.

Appendix

5.2 Features tracked

This is all the features tracked. Ordered by time of implementation during the project. As many features as possible were implemented during the time frame of the project. Features that aren't important doesn't change the result thanks to the feature weighting and clustering method used.

- KILL
- DEATH
- FLASH_THROWN
- SMOKE_THROWN
- GRENADE_THROWN
- MOLOTOV_THROWN
- STEP
- CROUCH
- FIRST_KILL
- SMG_FRAG
- SHOTGUN_FRAG
- MACHINEGUN_FRAG
- RIFLE_FRAG
- SNIPER_FRAG
- PISTOL_FRAG

- TRADE_KILL
- GRENADE_DAMAGE
- SITE_KILL
- T_ENTRY_KILL
- CT_ENTRY_KILL
- MID_KILL
- SITE_SPENT
- T_ENTRY_SPENT
- CT_ENTRY_SPENT
- MID_SPENT
- ENEMY_DURATION_FLASHED
- TEAM_DURATION_FLASHED
- DURATION_FLASHED
- FLASH_SUCCESSFUL
- PISTOL_ROUND_KILL
- POST_PLANT_KILL
- ALONE_KILL
- ALONE_SPENT
- EQUIPMENT_DIF_DEATH
- EQUIPMENT_DIF_KILL

- CROSSHAIR_MOVE_KILL_X
- CROSSHAIR_MOVE_KILL_Y
- TIME_OF_KILL
- TIME_OF_DEATH
- ALONE_DEATH
- AMOUNT_OF_MONEY
- KILL_FROM_BEHIND
- KILLED_FROM_BEHIND
- UNUSED_EQUIPMENT
- ENTRY_FRAG
- ENTRY_FRAG_FORCE
- ENTRY_FRAG_ECO
- TRADE_KILL_ECO
- TRADE_KILL_FORCE
- FIRST_KILL_FORCE
- FIRST_KILL_ECO